



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Scalable I/O Systems via Node-Local Storage: Approaching 1 TB/sec File I/O

A. Moody, G. Bronevetsky

May 21, 2008

Supercomputing
Austin, TX, United States
November 15, 2008 through November 21, 2008

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Scalable I/O Systems via Node-Local Storage: Approaching 1 TB/sec File I/O

Adam Moody and Greg Bronevetsky
Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory,
Livermore, CA 94551, USA
moody20@llnl.gov and greg@bronevetsky.com

Abstract

The growth in the computational capability of modern supercomputing systems has been accompanied by corresponding increases in CPU count, total RAM, and total storage capacity. Indeed, systems such as BlueGene/L [3], BlueGene/P, Ranger, and the Cray XT series have grown to more than 100k processors, with 100 TeraBytes of RAM and are attached to multi-PetaByte storage systems. However, as part of this design evolution, large supercomputers have lost node-local storage elements, such as disks. While this decision was motivated by important considerations like overall system reliability, it also resulted in these systems losing a key level in their memory hierarchy, with nothing to fill the gap between local RAM and the parallel file system.

While today's large supercomputers are typically attached to fast parallel file systems, which provide tens of GBs/s of I/O bandwidth, the computational capacity has grown much faster than the storage bandwidth capacity. As such, these machines are now provided with much less than 1GB/s of I/O bandwidth per TeraFlop of compute power, which is below the generally accepted limit required for a well-balanced system [8] [16]. The result is that today's limited I/O bandwidth is choking the capabilities of modern supercomputers, specifically in terms of limiting their working sets and making fault tolerance techniques, such as checkpointing, prohibitively expensive.

This paper presents an alternative system design oriented on using node-local storage to improve aggregate system I/O bandwidth. We focus on the checkpointing use-case and present an experimental evaluation of SCR, a new checkpointing library that makes use of node-local storage to significantly improve the performance of checkpointing on large-scale supercomputers. Experiments show that SCR achieves unprecedented write speeds, reaching 700GB/s on 8,752 processors. Our results scale such that we expect a similarly structured system consisting of 12,500 processors to achieve aggregate I/O bandwidth of 1 TB/s.

1 Introduction

As modern supercomputing systems approach PetaFlop performance, they set new records for processor counts and storage size. Together, the top 10 supercomputers in November of 2007 [1] contained 467,561 processors and hundreds of TBs of RAM. Upcoming systems such as the BlueGene/P at Argonne National Laboratory and Ranger at Texas Advanced Computing Center will push these limits even further. The storage system is an important component of any supercomputer since it serves as the machine's primary interface to the external world, providing it with input data and storing its intermediate and final results. Modern large-scale applications place great demands on storage systems, with typical problems taking up many TBs of space. While storage systems have successfully scaled in size with the largest supercomputers, reaching multiple PBs in capacity, the bandwidth they provide has not done the same. In particular, although 1GB/s of I/O bandwidth per 1TFlop of computing power is typically considered key to a well-balanced system, modern systems like the BlueGene/L at Lawrence Livermore National Laboratory and the upcoming BlueGene/P at Argonne National Laboratory achieve less than a tenth of that rate [14] [8].

This poor I/O bandwidth has a significant negative impact on the performance of modern applications. One use-case that is especially affected by this is checkpointing. As modern systems grow larger and more complex, they also grow less reliable, with many applications encountering mean times between failures on the order of hours or days due to hardware breakdowns [17] and soft errors [11]. For example, the BlueGene/L at Lawrence Livermore National Laboratory sees an L1 cache bit error every 3-4 hours and a hard failure every 7-10 days. Applications typically survive such failures by regularly checkpointing their state to stable storage and reloading this state upon a failure. Unfortunately, since checkpointing involves sending large fractions of system RAM state to the parallel file system, this is becoming increasingly expensive as systems grow ever-larger. In particular, dumping all of RAM on a 128K-processor BlueGene/L supercomputer to its parallel file system takes approximately 20 minutes [14] and the goal for the upcoming BlueGene/P at Argonne National Laboratory is a 30 minute full system checkpoint [8]. Thus, as supercomputers continue to grow in size, checkpointing will become both more critical and less practical, forcing PetaFlop-scale applications to either spend most of their time writing checkpoints or to use redundancy-based approaches that have overheads of more than 100%.

Another use-case for high I/O bandwidth is data-intensive supercomputing [4], which is an application domain focused on analyzing large data sets. This includes a variety of applications in biology, dynamic data-driven application systems [6], and large-graph analyses like those performed by Google and the intelligence community. These applications are special in that they analyze data from very large data sets, such as the GeneExpression database, which is expected to grow to multiple PBs in size [7] and data from the planned experiments on the Large Hadron Collider at CERN (ATLAS, CMS, ALICE and LHCb), which are expected to produce 25PBs per year. Altogether, the total amount of global electronic information is expected to continue to increase at a rate of 60% per year, which pushes future problem sizes even further. The fact that computational and storage capability of large supercomputers has thus far kept up with these growing problem sizes makes these machines very attractive for this application domain. However, the lack of I/O bandwidth between the processors and the storage currently makes these machines inadequate for this task [4].

There are two reasons for the relative slowness of today's I/O system designs. The first is that the storage systems used in modern large scale systems are designed to be separate from the main compute nodes. This ensures that the data is available to multiple machines and remains available if any given machine goes down. However, because this design puts the storage system on a separate network from the compute nodes, it also limits the available I/O bandwidth, with today's systems typically providing a few tens of GB/s. Further, since the storage system is a shared resource by design, the contention for this resource further reduces its effective bandwidth. The second reason is that modern parallel file systems are designed to provide a generic POSIX API that most users are accustomed to in their daily work. As such, they must provide various services, such as meta-data management, that are not needed in many simpler contexts such as checkpointing, and cause sub-optimal performance.

This paper presents a new scalable I/O system design that overcomes the limitations of modern high-performance I/O systems. The main insight of this design is that in large supercomputers node-local storage (storage elements installed on or near each compute node; DRAM, Flash, disk, etc.) has scaled much better than traditional parallel file systems. As such, we propose to use such storage elements as an extra level of cache between compute node memory and the parallel file system, providing any necessary consistency and reliability guarantees in software. We support this design direction by presenting SCR, a new checkpointing library that provides a highly scalable storage abstraction for checkpoint/restart applications. SCR caches checkpoint files in the node-local storage to achieve significant improvements in aggregate checkpoint bandwidth. To deal with reliability issues inherent to node-local storage, SCR implements several redundancy schemes that trade off performance with reliability and required storage space.

We evaluate the proposed I/O system design by experimentally evaluating SCR on Atlas and Thunder, two large-scale clusters at Lawrence Livermore National Laboratory. Each machine consists of more than 1,000 compute nodes with tens of TBs of RAM and delivers tens of TeraFlops of compute power. Both machines contain DRAM node-local storage via RamDisc, and Thunder also contains disk node-local storage. These two machines provide a unique large-scale testbed on which to evaluate the scalability and performance of this approach. Our experimental results show that I/O bandwidth scales to 700GB/s peak bandwidth and will reach as high as 1TB/s for an Atlas-style machine consisting of 12,500 processors.

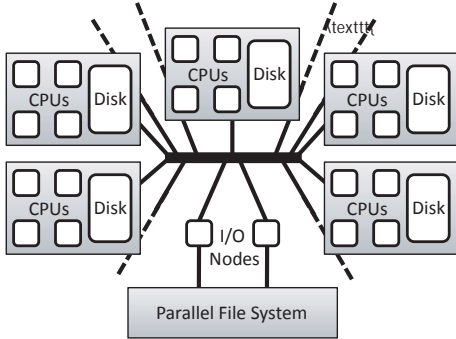


Figure 1: Traditional Cluster Design

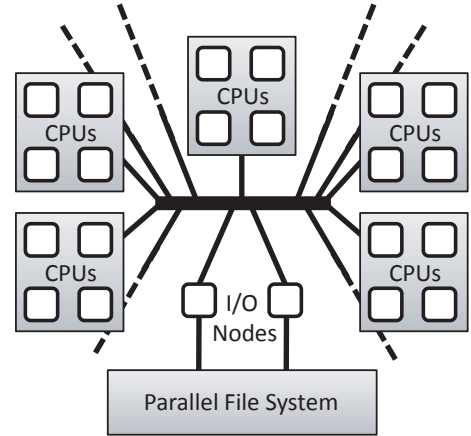


Figure 2: Modern Large Supercomputer Design

2 Current State of the Art

The typical design for today’s smaller clusters and the older generation of large supercomputers is shown in Figure 1. These machines are based on large numbers of multi-socket motherboards, each with a local disk. Since each node has access to a dedicated storage element and the number of such elements grows linearly with the size of the system, applications that run on such machines are guaranteed high I/O bandwidth that scales with the number of nodes in the system. Unfortunately, this design also has major drawbacks in the context of very large supercomputers. Since hard drives are tightly integrated into their host nodes in the traditional design, the failure of a hard drive results in the failure of an entire compute node and the application itself. Since hard drives are much less reliable than other system components [15], the mean time before failure (MTBF) of any large system constructed with integrated hard drives is unacceptably low. Another problem with the traditional design is the common practice of storing the OS image on each node’s hard drive and booting each node from this image. This approach complicates system maintenance, making common system update operations slow and invasive to system users. Finally, as most local disks were made available via relatively low-level APIs, such as the `/tmp` directory, they were generally left unused by application developers, significantly lowering their utility.

As a result of the above issues, modern large-scale machines like BlueGene/L, Bluegene/P, Ranger, and the Cray XT series all follow the design in Figure 2. Compute nodes have no local storage besides RAM, and all I/O funnels directly to the parallel file system through relatively thin connections. For example, most machines at Lawrence Livermore National Laboratory connect to the parallel file systems via a small number of 10Gigabit/s Ethernet connections. In contrast, the compute networks on these same clusters offer aggregate bandwidths on the order of TBs/s. The strong points of this modern design are its high reliability and manageability and the fact that application data is simultaneously available to multiple machines via the shared parallel file system. However, the corresponding cost is that these systems provide relatively poor bandwidth to the storage system when compared to their compute power and compute network bandwidth. With typical large supercomputers taking tens of minutes to transfer their RAM to the parallel file system, it is clear that even as modern large-scale systems are reaching new heights of performance, their designs are leaving behind critical aspects system performance, leading to significant shortcomings in their capabilities. This point is also noted by other researchers [4] [16].

One way of looking at the difference between traditional designs and the current state of the art is to think of local disks as an extra level of cache between compute node RAM and the parallel file system. In spite of the various good reasons to remove this cache, modern designs are now suffering from the resulting sparse memory hierarchy that contains a large gap between two levels: local RAM and the parallel file system. As such, the simplest way to overcome the limitations of current designs is to replace this level of the memory hierarchy without hitting the limitations of traditional designs. The main idea of our proposal is that compute nodes should be augmented with additional storage elements while ensuring that the following principles are followed:

- storage elements may fail but the computation must continue,

- the OS should be booted from a centralized repository and not from the local caches, and
- the caches must not be explicitly exposed to users and should instead be used by library implementors (ex: checkpointing libraries, virtual memory) to transparently improve application performance

Although modern large-scale system designs follow these basic principles, they consistently use expensive RAM to implement all node-local storage. By showing the utility of the node-local storage concept, we hope to popularize its use, thus motivating future system designs to supply more such memory perhaps using more cost-effective storage technologies such as Flash, MEMS, or disk.

In the rest of the paper we argue for the benefits of the above design by presenting SCR, a library for efficiently storing checkpoint data in node-local storage. Our evaluations of SCR on two large-scale supercomputers, the 9216-processor Atlas and the 4096-processor Thunder machines at Lawrence Livermore National Laboratory (currently the 29th and 47th largest supercomputers in the world), show the benefits of using node-local storage to improve checkpoint bandwidth and support our contention that such storage is critical to scalable supercomputer design. Section 3 discusses the basic problem of checkpointing and prior related work on scalable checkpoint storage. Section 4 describes the key algorithms implemented in SCR and how they impact performance and reliability. These algorithms are experimentally evaluated Section 5, proving the efficacy of our approach. Section 6 then describes how SCR addresses the needs of today’s large-scale platforms and the applications that run on them.

3 Checkpointing

Prior work on checkpoint storage has focused in two directions: distributed techniques that rely on node-local storage and more centralized techniques that focus on high-performance parallel file systems. The former category is best exemplified by work on diskless checkpointing by Plank, Li and Puening [13]. The primary idea of diskless checkpointing is to store most checkpoints in each node’s RAM, using replication or error correcting codes to guard against data loss in case of failure. Such low cost checkpoints could be taken frequently, with rarer and slower checkpoints to the shared file system. This idea was subsequently extended to local disks [12] and experimentally evaluated on small clusters [18] [9] and large SIMD machines [5].

More centralized file systems have also seen a significant amount of work. In particular, the open-source Lustre parallel file system [2] has been very effective. However, while it is routinely used in production, high-performance computing environments, the current limitations on the I/O bandwidth in such environments shows that Lustre alone cannot solve the problem. The Zoid I/O forwarding infrastructure [8] is designed to improve the I/O bandwidth of BlueGene/L and BlueGene/P supercomputers by optimizing various portions of the I/O software stack. However, while Zoid approaches the physical limitations of the BlueGene I/O subsystem, it cannot not overcome the fundamental I/O bandwidth bottleneck imposed by its design. Finally, Zest [19] presents a novel hardware/software approach for providing an high-quality, cost-efficient I/O system (high MB/sec/\$). However, this system is designed to work outside the compute network and thus far has only been shown to reach bandwidth of 800MB/sec. In contrast, our experiments show SCR reaching 700GB/sec of I/O bandwidth.

4 Algorithms

The Scalable Checkpoint / Restart (SCR) library is a library we designed to use node-local storage to implement a scalable I/O system to store checkpoint files. Any type of node-local storage can be used for this purpose. In Section 5, we evaluate DRAM and hard disk drives in particular, since those are the devices available on our large-scale clusters. The SCR library currently implements three different redundancy algorithms which tradeoff performance, storage requirements, and reliability:

- **Local** - checkpoint files are written to local storage
- **Partner** - checkpoint files are written to local storage, and redundantly copied to local storage on a partner node

- **XOR** - checkpoint files are written to local storage, and an XOR parity file of several checkpoints is computed and stored redundantly in the local storage of several nodes

4.1 Local

In **Local**, the library simply writes checkpoint files to storage on the local node. As such, it requires sufficient local storage to write the maximum checkpoint file size. This scheme is very fast, and it can withstand all failures that kill the application process but leave the node accessible. This failure class includes application bugs, such as segmentation faults or memory leaks, as well as, communication or file I/O errors that abort the application but leave the rest of the system intact. Most MPI errors fall under this classification. However, this scheme is susceptible to any failure that renders the node inaccessible, such as when the node loses power or its network connection.

4.2 Partner

In **Partner**, the library writes checkpoint files to storage on the local node, and it also copies files to storage on a partner node. This scheme is slower and requires twice the storage space as **Local**, but it can withstand failures in which a node is lost. In fact, it can withstand failures of multiple nodes, so long as a given node and its partner do not fail at the same time. In the current implementation, nodes are arranged in a ring that is based on the network topology, and each node selects the node D hops to the right to be its partner. As we show in the next section, a value of $D=1$ provides the best performance, because this often amounts to picking partners that are physically close to one another in the network, which reduces network contention. However, nodes located nearby each other may have some reasonably high chance of failing at the same time, such as when a common network switch fails or a power breaker feeding a section of the cluster shuts off. The hop distance, D , can be increased to handle such failures, but this may also increase the cost to checkpoint.

4.3 XOR

In **XOR**, nodes are first assigned to disjoint sets, each of size N . Once these sets are defined, the library writes checkpoint files to storage on the local node. Then, all nodes in the same set compute a byte-wise XOR of their checkpoint files. The resulting XOR parity file is split into N equal-sized segments, and each node is assigned as the owner of one segment. Finally, each node writes its XOR segment to local storage, and it also copies its segment to storage on a partner node in the XOR set. SCR currently uses a ring-based, reduce-scatter algorithm to compute and split the XOR file. Thus, in this scheme, checkpoint files are not duplicated, but the XOR file is.

This scheme can withstand multiple node failures as long as two nodes from the same XOR set do not fail at the same time. If a node fails, its checkpoint file can be reconstructed using the remaining $N - 1$ checkpoint files and the XOR parity file. This algorithm takes more time than **Partner**, but it requires less storage space. Whereas **Partner** must store two full checkpoint files, **XOR** stores 1 full checkpoint plus two XOR file segments, where the segment size is roughly $1/N$ th the size of a checkpoint file. The XOR segments can be made as small as desired by increasing the XOR set size, N . However, increasing the XOR set size also increases the chances that more than one node in the set will fail at the same time.

5 Experiments

We conducted experiments on two production clusters at Lawrence Livermore National Laboratory: Atlas and Thunder. Atlas consists of 1152 nodes, where each node contains four dual-core AMD Opteron processors and 16GB of main memory. Nodes are connected via Infiniband 4x links running at DDR, which provides a peak MPI bandwidth of 1.5GB/s. Atlas nodes have no local hard drives. Thunder consists of 1033 nodes, where each node contains four single-core Intel Itanium2 processors and 8GB of main memory. Nodes are connected via the Quadrics Elan4 interconnect, which provides approximately 900MB/s peak MPI bandwidth. Thunder nodes have a local hard drive. Both machines use a fat-tree network topology, where individual nodes are connected to lower-level switches, which are then connected to multiple higher-level switches. The Infiniband network for Atlas employs static routing,

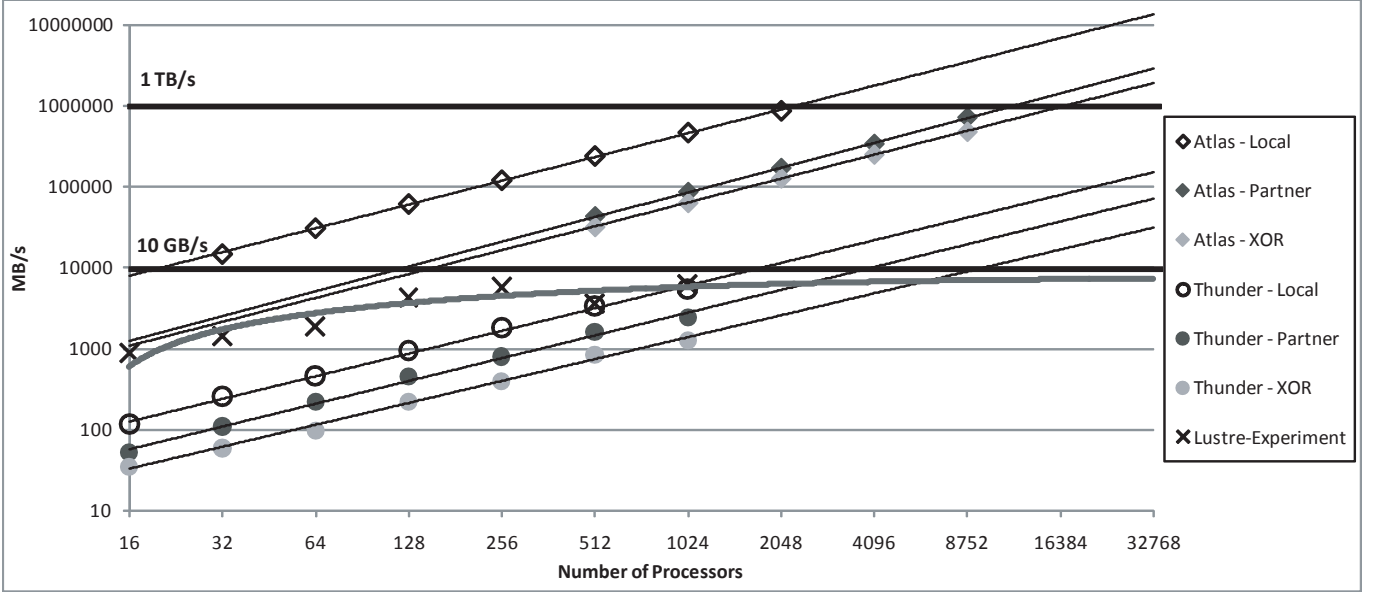


Figure 3: Aggregate Bandwidth

while the Quadrics network on Thunder provides dynamic routing. On Atlas, checkpoint files were written to the RamDisk (a file system maintained in DRAM), and on Thunder, they were written to the local hard drive. In some places the graphs are missing specific data points due to our limited access to the target machines. In all such cases we focused on getting the largest-scale data points possible; the final paper will include all data points.

5.1 Major Results

We begin by presenting the major results of this study. Figure 3 shows the aggregate bandwidth achieved by all the algorithms: **Local**, **Partner**, and **XOR** on Atlas/RamDisk and Thunder/disk. Processor counts, which scale from 16 to 32k, are shown on the x-axis and the bandwidth, in MB/s, is plotted on the y-axis. To provide context, the graph also includes two lines that correspond to the Lustre parallel file system which is attached to both clusters as a shared resource. **Lustre-Experiment** corresponds to experimental bandwidths achieved by having increasing numbers of Thunder nodes save their entire memories to Lustre (best of 10 trials). Furthermore, the 10GB/s bandwidth line corresponds to the current peak bandwidth that this parallel file system can provide.

Each experimental line is augmented with a trend-line that predicts the performance of each configuration for processor counts where data is not currently available. We applied exponential fit functions for all configurations except for **Lustre-Experiment**, where a 2nd order polynomial produced a better fit to the data. We use these trends to make predictions about larger systems built from the same architecture.

The major conclusion to be made from Figure 3 is that the use of node-local storage for checkpointing is far more scalable than using the parallel file system. Even if we assume that Lustre delivers its peak bandwidth for all processor counts, its 10GB/s is quickly overtaken by all of the Atlas configurations. In particular, this data allows us to predict that node-local storage can be used to achieve 1TB/s of aggregate I/O bandwidth on an Atlas-type system with 2,380 processors using the **Local** algorithm and 12,500 processors using the more reliable **Partner** algorithm.

Thunder’s disk-based performance is lower than Lustre for smaller processor counts. Nevertheless, the superior scalability of node-local storage means that Thunder-**Local** will beat Lustre’s peak bandwidth at approximately 1,890 processors and Thunder-**Partner** will beat Lustre at approximately 4,270 processors. Furthermore, Thunder-**Local** exceeds Lustre’s real-world bandwidth at much smaller processor counts. The important point to be made is that node-local storage provides natural scalability. Regardless of the choice of storage technology, the aggregate I/O bandwidth will naturally scale with overall system size.

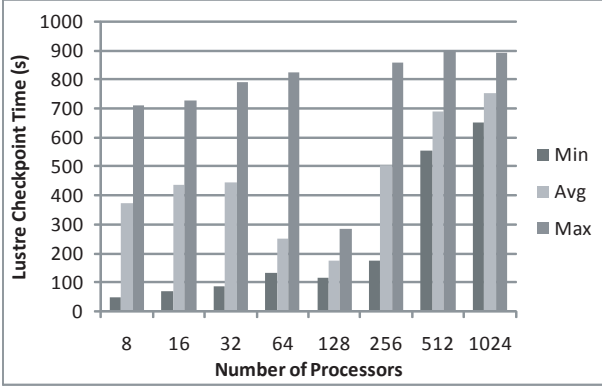


Figure 4: Time to take full checkpoints to Lustre (range of 10 trials)

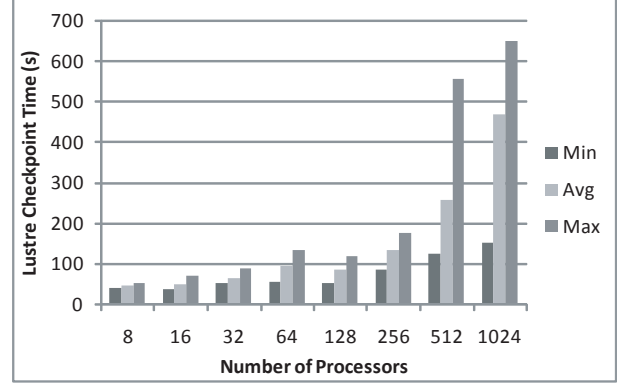


Figure 5: Time to take full checkpoints to Lustre (minimum of 10 trials, range across processors)

Figure 3 shows clear differences between the performance characteristics of the three checkpoint storage algorithms, with **Local** outperforming **Partner**, which outperforms **XOR**. This is not surprising, since each algorithm adds additional communication and computation on top of the previous one. These results can be used to intelligently trade off performance, reliability, and storage space requirements for saving checkpoint files. In particular, the fact that hard disk drives are much cheaper per GB than DRAM means that the **XOR** algorithm, which requires less space at the cost of increased time, is less useful for disks than it is for DRAM.

We explore the performance characteristics of the parallel file system in Figure 4, which shows the full range of checkpoint times across the 10 runs in each Lustre experiment run on Thunder. It shows that although the minimum checkpoint times are low (the minimum times are used in Figure 3), there is a large difference between the minimum and the maximum times, with the average being much higher than the minimum. For small processor counts, the average and maximum times are, respectively, 626% and 1280% higher than the minimum time. This difference drops to 15% and 37%, respectively, for large processor counts. Furthermore, Figure 5 presents the minimum, average and maximum checkpoint times across all processors within the minimum-time checkpoints for each processor count. The same pattern is seen as before, except that in this case, the differences rise with increasing processor count. The average and maximum are respectively 12% and 33% higher than the minimum for small processor counts. This degrades to a respective 211% and 332% for larger processor counts. This data shows the inherent variability of accessing a shared resource such as the parallel file system, which is in use simultaneously by multiple compute nodes from multiple applications. This variability creates a significant cost for large-scale, tightly synchronized applications which are quite sensitive to even smaller timing effects such as Operating System noise [10]. While node-local storage is not immune to timing variation effects due to slow nodes, these effects are already well-known and can be overcome by (i) a judicious choice of nodes on which to run the application or (ii) alternative storage algorithms that balance the load across fast and slow nodes.

5.2 Local

Figure 6 shows the per-node bandwidth of the **Local** algorithm on 256 nodes on Atlas and 256 nodes on Thunder. The x-axis corresponds to the number of processes writing data on each node. The left y-axis shows the bandwidth for Atlas, while the right y-axis shows the bandwidth for Thunder. Both y-axes are in units of MB/s. Each bar corresponds to experiments with different checkpoint sizes per writer process. On Atlas, the per-node bandwidth increases with increasing number of writer processes per node. There are two explanations for this. First, on Atlas nodes, memory and processor sockets are configured in a NUMA architecture, such that each socket is connected directly to a local bank of memory. This enables different processors to access memory banks in a contention-free manner. In addition, the implementation may be benefiting from pipelining of file I/O requests in the operating system. Regardless, while each processor accesses DRAM at very high bandwidths, it is clear that the processors collectively have not yet saturated the available DRAM bandwidth on the node.

The per-node bandwidth trend moves the opposite direction on Thunder. On this system all processes on a

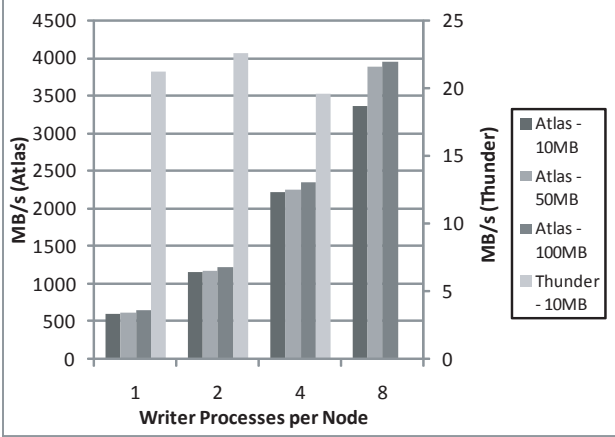


Figure 6: Per-node bandwidth with *Local* (256 nodes on Atlas, 256 nodes on Thunder)

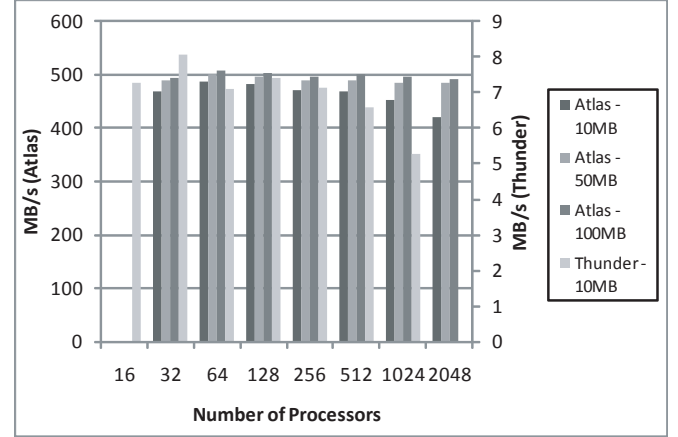


Figure 7: Per-processor bandwidth with *Local* - Scaling

node share a single hard drive. In the current implementation, the processes compete with each other for access to the drive, and this contention decreases the aggregate bandwidth. A better implementation could reduce this effect by scheduling access to the drive in order to maintain full bandwidth.

Figure 7 shows how the *Local* algorithm per-processor bandwidth scales with increasing processor counts. In each case we used the optimal number of writer processes per node: 8 for Atlas and 1 for Thunder. The performance scales very well with increasing processor counts on both clusters. This scaling is nearly perfect on Atlas, while the Thunder results show some fall-off. In our experiments, we found that there is some spread in hard drive write speeds on different nodes. Due to synchronization in the implementation, one slow node acts to slow down the entire operation. As more nodes are used, the likelihood of running with such slow nodes increases, and thus the performance falls off with more processors. While we have attempted to filter out some of these nodes, this is nevertheless a real and well-known phenomenon in large-scale systems. Even so, the Thunder results scale very well and remain in a range of 5 to 8MB/s per processor for all processor counts tested.

5.3 Partner

For *Partner*, we begin by analyzing how the hop distance, D , and the number of writing processes per node influence overall node bandwidth. Figures 8 and 9 show the per-node *Partner* bandwidth for Atlas and Thunder, respectively. The bandwidth is shown in MB/s along the y-axis, and the partner hop distance, D , which varies from 1 to 32 is shown along the x-axis. For each value of the hop distance, we show the measured bandwidth for a set of different numbers of writing processes per node. We varied the number of processes per node from 1 to 8 in the Atlas runs and from 1 to 4 in the Thunder runs.

First, in Figure 9, note how the per-node bandwidth generally increases as the number of processes per node is increased on Atlas. This is due to a pipelining effect. While transferring a file, a process reads a chunk of file from storage, and then sends it to its partner. It receives an incoming chunk from another process simultaneously with its send, as the network links are bidirectional. Once the incoming chunk is received, it is written to storage. The next outgoing chunk is then read from storage, and the cycle is repeated until the entire file has been transferred. The file chunk size is large enough to achieve the peak MPI link speed. However, it takes time for a process to read and write these chunks in storage, during which time, the process does not utilize the link. By adding more processes per node, processes can pipeline their network operations and keep the link better utilized. The net result is that the per-node bandwidth increases as more processes run on the node.

Second, also in Figure 9, note how this pipelining effect saturates as the hop distance increases from $D = 1$ to $D = 32$. Since higher values of D encounter more network contention (discussed below), there is less link bandwidth available to be pipelined.

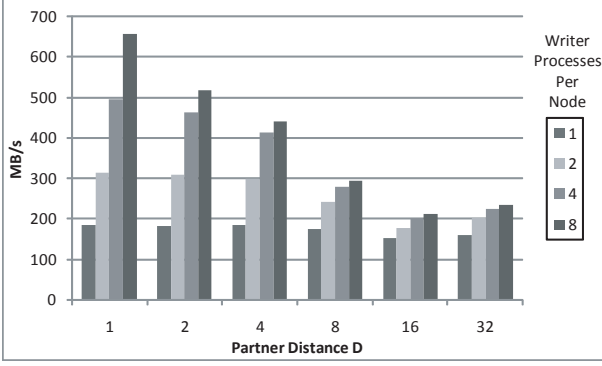


Figure 8: Per-node bandwidth on Atlas with **Partner** (1094 nodes)

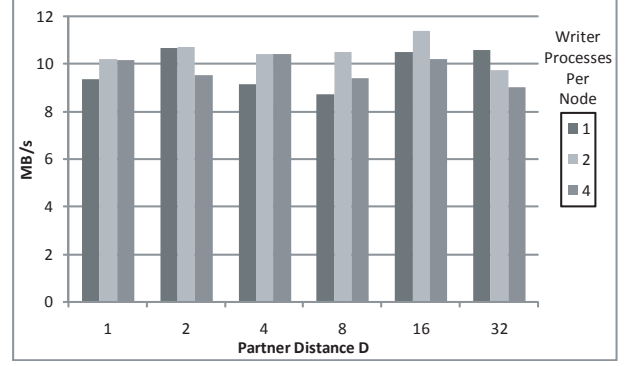


Figure 9: Per-node bandwidth on Thunder with **Partner** (256 nodes)

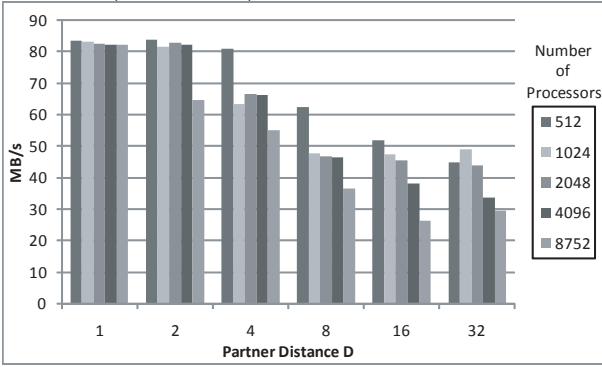


Figure 10: Per-processor bandwidth on Atlas with **Partner** (8 processes per node)

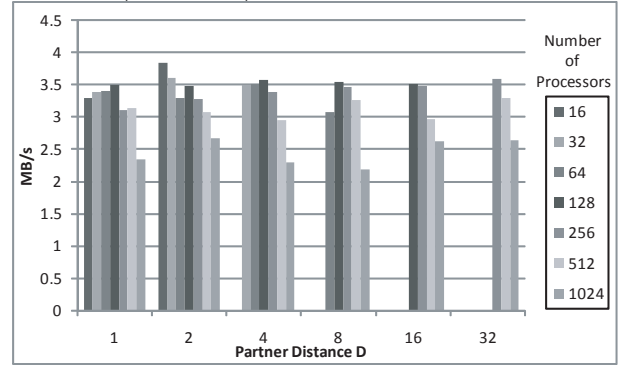


Figure 11: Per-processor bandwidth on Thunder with **Partner** (1 process per node)

In contrast, in Figure 9, Thunder’s per-node bandwidth remains basically constant with increasing processes per node. In this case, the hard drive, which is much slower than the network link speed, is a bottleneck. A single process is capable of saturating the disk bandwidth. When running with more processes per node, the disk bandwidth is split equally among them.

Figures 10 and 11 show the per-processor **Partner** bandwidth for Atlas and Thunder, respectively. We scale the number of processors from 512 to 8752 on Atlas and from 16 to 1024 on Thunder. We show results when using the number of processes per node which maximizes the per-node aggregate bandwidth on both systems: 8 for Atlas and 1 for Thunder. The per-processor bandwidth is shown in MB/s along the y-axis, and the partner hop distance, D , which varies from 1 to 32 is shown along the x-axis.

In Figure 10, the first trend to note is how the bandwidth on Atlas generally falls off with increasing hop distance. This effect is clear as the per-processor bandwidth steadily drops from 80MB/s when $D = 1$ to less than 50MB/s when $D = 32$ for a given number of processors. This effect is caused by increased network contention. When D is small, many partner nodes are located on the same leaf-level switches as their sending nodes. In this case, packets bounce off of the first-level switch and are forwarded immediately to their destination without contention. However, once D grows larger, partner nodes are located on different leaf-level switches. Packets must contend with each other in higher-level links and switches. Due to the static routing in Infiniband networks, this contention can be severe, and this leads to the fall off in **Partner** bandwidth.

Note that this trend is not seen in the Thunder results, in Figure 11, because its Quadrics network is dynamically routed, which avoids hot-spots.

The second trend to note, in Figure 10, is the scalability of the **Partner** bandwidth as the number of processors is increased for a fixed value of D . For low values of D , like $D = 1$ or $D = 2$, the bandwidth scales almost perfectly. For $D = 1$, the bandwidth drops by only 1.6% from 83.5MB/s to 82.2MB/s as the number of processors increases from 512 to 8752. For $D = 2$, the results are almost as scalable, until there is a more significant drop in performance

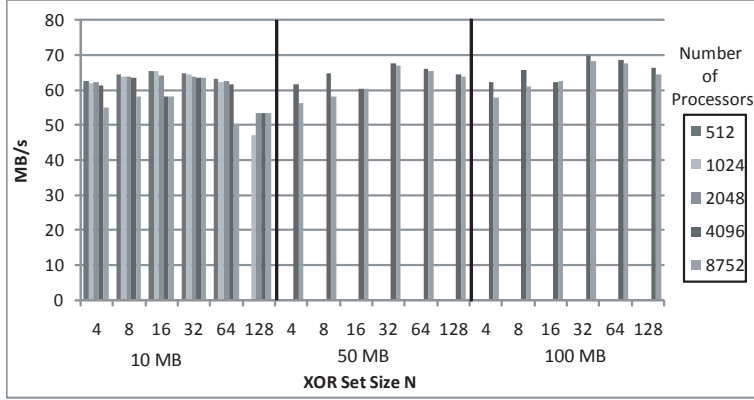


Figure 12: Per-processor bandwidth on Atlas with XOR (8 processes per node)

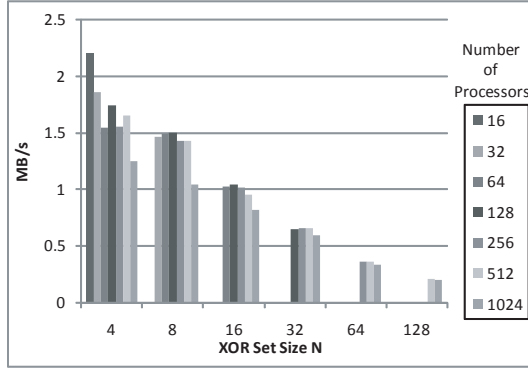


Figure 13: Per-processor bandwidth on Thunder with XOR (1 process per node)

for the largest processor count of 8752. For larger values of D , scalability falls off for smaller processor counts. This is due, again, to network contention, since using more processors leads to more packets flowing through the same network. Ultimately, although the scaling is not perfect for larger values of D , the **Partner** bandwidth still scales reasonably well even for values as large as $D = 32$, where performance remains in a range between 30 to 50MB/s.

The results for Thunder, in Figure 11, also scale quite well. Although, there is some fall off in performance for very large process counts. This effect is due to the increased likelihood of running with slow nodes as discussed in Section 5.2.

5.4 XOR

Figures 12 and 13 show the per-process XOR bandwidth for Atlas and Thunder, respectively. We used the optimal number of processes per node on both systems: 8 for Atlas and 1 for Thunder. The per-processor bandwidth is shown in MB/s along the y-axis, and the XOR set size, N , is shown along the x-axis. In Figure 12 for Atlas, we also test with three different file sizes of 10, 50 and 100MB, which are shown along the x-axis. For each value of the XOR set size, we show the measured bandwidth for a set of processor counts ranging from 512 to 8752 on Atlas and from 16 to 1024 on Thunder.

Like the **Partner** algorithm, the XOR algorithm scales very well as the number of processors increase. This can be seen in Figure 12 for an XOR set size of $N = 4$ and file size of 10MB. The per-process bandwidth only falls by 1.9% from 62.6MB/s to 61.4MB/s as the number of processors increases from 512 to 4096. This scalability trend holds generally for the different XOR set sizes and different file sizes.

One may note, in Figure 12, that performance is scalable but relatively lower for XOR set size $N = 128$ and a file size of 10MB. For this file size, the XOR segments are only 80KB, which is less than the file chunk size of

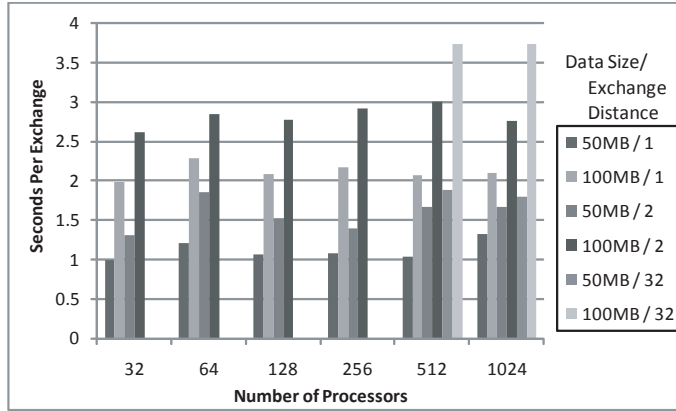


Figure 14: Time to exchange data between all processes

128KB. This reduces the benefit of pipelining. Note that this performance drop off is not as significant for file sizes of 50MB or 100MB, where segment sizes are larger.

On Thunder, the clear trend, in Figure 13, is the reduced performance with increasing XOR set size. A larger XOR set size, leads to a higher number of segments, each of which is smaller for a given checkpoint file size. This leads to more individual disk operations of a smaller size, which reduces performance.

In a head-to-head comparison on 2048 processors on Atlas with 1 process per node, the best XOR per-process bandwidth achieves 109MB/s which is 57% of the best **Partner** bandwidth of 190MB/s. With 8 processes per node, the best XOR per-process bandwidth is 64MB/s, which is 78% of the best **Partner** bandwidth of 82MB/s. Thus, the XOR scheme gains a bigger advantage from pipelining effects, which help to overlap the additional XOR computations.

On Thunder, this trend does not hold; XOR performance is mostly independent of the number of processes per node. In general, the recommendation is to use **Partner** when writing to a hard drive like on Thunder. The primary benefit of using XOR is to conserve storage space, which is of much less concern when writing to a hard drive as opposed to DRAM.

5.5 Full Exchange

Checkpointing has a number of important applications beyond fault tolerance. One example is to migrate processes from processor to processor to either improve the mapping of the application’s virtual communication topology to the underlying physical topology or to move away from poorly-performing nodes. This operation would be prohibitively expensive using conventional storage systems because it would involve shipping the checkpoints of each process to the parallel file system and then sending them all back to their new locations. In contrast, node-local storage makes this task very efficient. Figure 14 shows the number of seconds required for all processors to exchange checkpoints with all other processors on Atlas, as the processor count increases from 32 to 1024. The evaluations include processes sending either 50MB or 100MB files to partners that were distance 1, 2 or 32 away. The results show that for both file sizes the exchange operation scales linearly, with 50MB global exchanges taking between 1.0s and 1.8s and 100MB taking between 2.0s and 3.7s. Increases in distance cause exchange times to increase due to additional network contention. Given this performance, global migration becomes a viable operation during normal operation execution.

6 Experience in Production

The **pf3d** application team at Lawrence Livermore National Laboratory has been using the SCR library in daily production for the past seven months on Atlas. The predominant failure type they encounter is node loss, and since

they have sufficient spare memory, they opt to use the **Partner** algorithm. They cite at least four major benefits of using the SCR library:

- drastically faster checkpoint times,
- consistent checkpoint times,
- finer granularity to de-schedule a job, and
- drastically improved mean time before failure

The parallel file system on Atlas is Lustre, which with the current hardware, provides a peak bandwidth of about 10GB/s. For a job of 4096 processes on 512 nodes, the time to checkpoint **pf3d** to the parallel file system averages 3.5 minutes. However, as the parallel file system is a shared resource, this performance varies widely depending on the load from other jobs. In the worst case, the team timed one checkpoint which took 1.5 hours for 8192 processes on 1024 nodes. Due to this high cost and variability, the team has traditionally configured the application to write a checkpoint every 2 hours.

Since the **pf3d** team began using the SCR library, the time to write a checkpoint for 4096 processes on 512 nodes dropped to just over 4 seconds. Furthermore, because the checkpoint files are written to resources that are dedicated to the job, the performance is consistent and independent of other jobs. Since the checkpoint cost is so low, a checkpoint is now written on every time-step, which occurs every 15 minutes.

The actual checkpoint timing data for the 512 node runs as well as 128 node runs are shown in the table below.

Processes / Nodes	Filesize	SCR Time	Lustre Time	SCR BW	Lustre BW	Speedup
1024 processes / 128 nodes	308 MB	3.5 s	65 s	88 GB/s	4.7 GB/s	19x
4096 processes / 512 nodes	289 MB	4.5 s	208 s	257 GB/s	5.6 GB/s	46x

More frequent checkpointing enables a job to save its work more often and thus restart from a later point upon recovering from a failure. When checkpointing every 2 hours, on average, a job loses 1 hour of work for each failure. In contrast, 15-minute checkpoint intervals lead to an average of only 7.5 minutes being lost to a failure.

In addition to fault tolerance, another benefit of more frequent checkpoints is to better enable fair system allocation. Large-scale simulations may run for days or weeks at a time. To timeshare the system among multiple users, jobs are often scheduled into relatively small timeslots ranging anywhere from several hours to several days in length. Since timeslot de-scheduling is equivalent to failure, cheaper and more frequent checkpoints allow applications to get more useful work done during their time allocations.

Finally, the biggest benefit the **pf3d** team gained by switching to the SCR library was a significant reduction in the system’s mean time before failure, which improved dramatically from 1.5 hours to 1.5 days. The reason for this is that the most predominant failure encountered by the application is network errors. While **pf3d**’s use of SCR actually increases network traffic due to more frequent checkpoints, the network is used more efficiently. Data is passed from each node to its neighboring partner node (**pf3d** uses distance $D = 1$), and in most cases such packets bounce off of the first-level switch to their destination with no contention. Writing to the parallel file system, on the other hand, requires the data to pass across the network, which involves more switches and links and leads to more contention. Additionally, the parallel file system itself is exposed to less stress. Removing these two pressures from the system significantly reduced the failure frequency.

In all, the SCR library has substantially increased the efficiency of the system for large-scale, production runs of **pf3d**.

7 Conclusions

The goal of this paper is to present node-local storage as a scalable I/O system design for large scale supercomputers. Modern supercomputer designs, such as those used to construct BlueGene/L, BlueGene/P, Ranger, and the Cray XT series remove most node-local storage from compute nodes, forcing all storage I/O to funnel to an external parallel file system. While this approach simplifies system design and administration, and while it has some positive effect on reliability, it also creates an important bottleneck between the compute nodes and their storage. As a

result, modern supercomputers are limited in the types of operations they can efficiently perform. Checkpointing, which is becoming increasingly critical as supercomputers grow larger and less reliable, now takes up tens of minutes for large-scale applications. Furthermore, data-intensive supercomputing, a new and promising application domain, is currently beyond the capabilities of these machines because they cannot efficiently access and operate on large amounts of data.

This paper presents an experimental evaluation of using node-local storage to support checkpointing, one of the key use-cases. We describe SCR, a new checkpointing library that uses node-local storage to significantly improve the performance and scalability of checkpointing, and we use this library to experimentally validate our proposed design. Our experiments show that all of SCR's checkpointing algorithms scale extremely well to large numbers of processors, showing that we can reach 1TB/s for one of the algorithms using 2,380 processors and reach the same mark with a more reliable algorithm using 12,500 processors. This is in contrast to existing centralized storage technologies that currently reach a few tens of GB/s. Furthermore, this scaling behavior is consistent across different architectures, networks, and storage technologies, showing that node-local storage is a general and scalable approach to supercomputing storage in a wide variety of real-world environments. Further, we presented an evaluation of the scaling properties of the Lustre parallel file system and found that, although Lustre scales well upto its peak bandwidth, this peak does not itself scale with the size of the supercomputer because the parallel file system is not integrated with the compute nodes or the compute network. In addition, because Lustre is used as a shared resource, performance is erratic across runs and across different processes within the same run. This inconsistency results in reduced application performance and makes it harder for users to plan their batch runs.

Our ongoing work is focusing on extending our evaluation of scalable node-local storage to new application domains. In particular, we plan to extend the set of supported APIs from checkpointing to file I/O and virtual memory. These extensions will enable us to study the scalability and performance of node-local storage for these domains and enable us to make more specific recommendations for future supercomputer designs.

As supercomputing systems approach the PetaFlop performance range, their immense computational power must be balanced by scalable systems to connect these machines to the outside world. This paper provides large-scale experimental evidence that node-local storage is an effective scalable storage technology for future supercomputer designs.

References

- [1] <http://www.top500.org>.
- [2] <http://www.lustre.org>.
- [3] NR Adiga, G Almasi, GS Almasi, Y Aridor, R Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, AA Bright, and J. An Overview of the BlueGene/L Supercomputer. In *IEEE/ACM Supercomputing Conference*, 2002.
- [4] R. E. Bryant. Data-intensive supercomputing: The case for DISC. Technical Report CMU-CS-07-12, Carnegie Mellon University.
- [5] Tzi-Cker Chiueh and Peitao Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *International Symposium on Fault-Tolerant Computing*, 1998.
- [6] Frederica Darema. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. In *International Conference on Computational Science*, pages 662–669.
- [7] Tony Hey and Anne Trefethen. The data deluge: An e-science perspective. In *Grid Computing: Making the Global Infrastructure a Reality*, pages 809–824.
- [8] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.
- [9] Hai Jin and Kai Hwang. Distributed checkpointing on clusters with dynamic striping and staggering. In *Asian Computing Science Conference on Advances in Computing Science*, 2002.

- [10] Darren J. Kerbyson, Adolfo Hoisie, and Harvey J. Wasserman. Use of predictive performance modeling during large-scale system installation. *Parallel Processing Letters*, 15(4):387–396, 2005.
- [11] Sarah E. Michalak, Kevin W. Harris, Nicolas W. Hengartner, Bruce E. Takala, and Stephen A. Wender. Predicting the number of fatal soft errors in los alamos national laboratorys ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.
- [12] James S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *Symposium on Reliable Distributed Systems (SRDS)*, 1996.
- [13] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.
- [14] Rob Ross, Jose Moreira, Kim Cupps, and Wayne Preiffer. Parallel I/O on the IBM Blue Gene/L System. Technical report, BlueGene Consortium, 2005.
- [15] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *USENIX Conference on File and Storage Technologies*, 2007.
- [16] Bianca Schroeder and Garth Gibson. Understanding failure in petascale computers. In *USENIX Conference on File and Storage Technologies*, 2007.
- [17] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006.
- [18] Lus M. Silva and Joo Gabriel Silva. An experimental study about diskless checkpointing. In *EuroMicro*, 1998.
- [19] Nathan Stone, Doug Balog, Paul Nowoczynski, Jason Sommerfield, and Jared Yanovich. Zest: The maximum reliable TBytes/sec/\$ for petascale systems. In *Supercomputing Storage Challenge*, 2007.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.